HPC TOOLS

Task 2

Unai Iborra Albéniz

Introduction

The objective of this document is explaining the performance results of using different compiling optimizations to the previous task's Gauss solver code.

Row swapping

In the previous task 1 baseline code, while the application of the Gauss solver formula was algorithmically correct, had a low precision in matrices with a size bigger than 400. The imprecision in the results were due to accumulated floating point errors by dividing by small numbers in the division from the pivot. To help fix this imprecision, a row swapping feature has been applied to each iteration of the main loop.

It works by finding the row with the maximum absolute value in the iteration column, starting from the column index row, downwards:

```
size_t piv_row = col;
double max_val = fabs(a[m_idx(n, col, col)]);
for (size_t row = col + 1; row < n; row++)
{
	double val = fabs(a[m_idx(n, row, col)]);
	if (val > max_val)
	{
		max_val = val;
		piv_row = row;
	}
}
```

Then, if the maximum is not already in the actual column, the current row is swapped by the piv_row (the row with the maximum absolute value in the current main loop iteration column):

```
if (piv_row != col)
{
	for (size_t k = 0; k < n; k++)
	{
		double tmp = a[m_idx(n, col, k)];
		a[m_idx(n, col, k)] = a[m_idx(n, piv_row, k)];
		a[m_idx(n, piv_row, k)] = tmp;
	}
```

Finally, the b matrix columns are also swapped in order to mantain the coherency in the matrix equation:

```
        for (size_t j = 0; j < nrhs; j++)
        {
                double tmpb = b[col * nrhs + j];
                b[col * nrhs + j] = b[piv_row * nrhs + j];
                b[piv_row * nrhs + j] = tmpb;
        }
}
```

Thanks to this implementation, the Gauss solver divides with a bigger pivot, improving the numerical stability and reducing amplification floating point errors.

Tests of this implementation show results with much more precision than results without it. For example, matrices of 4096 where the results from the Lapacke dgesv implementation had big differences, has a correct result within an epsilon of 10^-4.

This implementation has been added inside a `#ifdef ROW_SWAPPING` block to allow compilation with and without the improvement in order to determine if the performance cost of this code is relevant.

Compilation

In order to facilitate the compilation job with all the optimization levels a python script has been created (/scripts/compile.py) to compile the code inside the FT3. It creates the following folder tree structure **/build/{compiler version}/{optimization level}/**, where the following files and folders are created:

1. “**dgesv**” and “**dgesv_rs**”: The compiled binaries. “dgesv” does not use the row swapping feature and dgesv_rs does. This will allow analyzing the overhead of the feature.
2. “**compile_logs.log**” and “**compile_logs_rs.log**”: They are the compilation logs of dgesv and dgesv_rs, saved in text files. Useful for analyzing where the compiler is vectorizing the code and where it is not.
3. “**asm**” folder: contains the compiled assembly codes of the dgesv and dgesv binaries using the -S flag. It will facilitate the analysis of the compilation optimizations.

The following flags have been used in the different optimization compilations:

```
no_vector_flags = "-fno-tree-vectorize -fno-tree-slp-vectorize"
vector_flags = "-march=native -ftree-vectorize -ftree-slp-vectorize"
vector_info_flags = "-fopt-info-vec -fopt-info-vec-missed -fopt-info-vec-all -fopt-info-vec-
optimized"
optimization_flags = {
        "O0": f"-O0 {no_vector_flags}",
        "O1": f"-O1 {no_vector_flags}",
        "O2": f"-O2 {no_vector_flags}",
        "O3": f"-O3 {vector_flags} {vector_info_flags}",
```

```
        "Ofast": f"-Ofast {vector_flags} {vector_info_flags}",
}
```

Benchmarking

Benchmarking with 64 threads

In order to benchmark the different binaries in an automatized way with a slurm job, the /scripts/benchmark_64c.py script was made. It uses a `ThreadPoolExecutor` that creates a thread pool with 64 threads in total. Each thread executes a binary 5 times, and when the execution finishes, the thread runs another binary from the pending binary queue. When the all the binaries are benchmarked, it creates the benchmark_results.json file with all the execution times from the different iterations (compiler versions, with or without row swapping and matrix sizes). The json result from running this script in an exclusive node is located in /results/raw/benchmark_results_64c.json.

After analyzing the results, the observation that the times were incredibly inconsistent was made. As an example, for the same matrix size, in the same optimization level and compiler (gcc-8.4.0, -O2, 2048 size matrix with row swapping) a result of 50,175ms and another of 210,473ms was obtained (~4.2x the execution time from the other execution). This inconsistencies were present all around most of the executions of the different binaries.

The hypothesis that using 64 threads at once for the benchmarking was the cause of the inconsistencies was made. Although running 64 threads reduces the total benchmarking time, it also led to slower independent executions, since all processes shared the same L3 cache, increasing cache misses. In addition, the FT3 processors have 32 physical cores with Intel hyper threading, meaning that 64 threads are not truly independent execution units, which likely contributed to the variability.

In order to prove the hypothesis and test the results of the benchmark between using 64 threads and 1 thread, the scripts/benchmark_variation.py script was made. It tested the gcc-8.4.0 in O2 optimization level with a matrix size of 2048 binary with 20 executions. The obtained results are located at /results/raw/benchmark_results_variation_single_thread.json. The results had significantly smaller time variations between executions (maximum of ~1.6x times from the fastest execution and the slowest execution within the 20 executions, probably due to not requesting the node as exclusive).

Another difference from the 64 and 1 thread benchmarks was that the individual time results were significantly faster in the single threaded execution. As an example, the fastest result using 64 threads (with the same binary) was 50,175ms, while it only took 22,688ms for the single threaded execution (~2.2 speedup for single threaded execution). The slowest result using 64 threads took 210,473ms and 37,090ms in single threaded (~5.7 speedup for single threaded execution).

The results from this test show that for obtaining coherent and trustworthy benchmarking results single threaded execution is much more logical and precise.

Benchmarking row swapping

With the intent of analyzing the cost of using the row swapping feature, a small benchmark between binaries with and without the feature was made.

The results from this benchmark showed no noticeable time differences between using row swapping and not using it. The times were sometimes slightly faster with row swapping and some others without it. The conclusion that because the row swapping code is located at the main loop and only executes N times it is not a relevant part of the final execution time was made.

Due to the performance of using row swapping being not noticeable, all the following benchmarks were done only using the compiled binaries with the row swapping feature.

Final benchmarking with single threaded execution

Finally a script for testing all the binaries with different compilers and compilation options with single threaded execution was made. The script can be found in /scripts/benchmark_1c.py and is a adaptation of the first script that used 64 threads, but only using one and executing all the binaries sequentially. It executes each binary five times in order to get a more precise result by using either the mean or median results of the iterations. In order to summarize all the iterations and results, the scripts /scripts/summarize_results.py and /scripts/create_tables.py scripts have been made.

The following tables present the median execution times (in seconds) obtained for each compiler and optimization level:

=== gcc_8_4_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	9.31	2.223	2.108	1.955	1.951	0.119
2048×2048	114.686	22.66	22.298	21.299	21.121	0.789
4096×4096	1116.5	300.566	298.174	331.365	255.419	5.73

=== gcc_10_1_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	9.061	2.261	2.071	1.838	1.89	0.117
2048×2048	87.058	28.581	33.467	30.195	30.83	0.816
4096×4096	1088.5	441.398	295.212	293.811	257.954	5.746

=== gcc_11_4_0 ===

Matrix Size	O0	O1	O2	O3	Ofast	Ref
1024×1024	9.103	2.364	2.236	1.895	1.834	0.139
2048×2048	85.231	33.015	36.077	33.543	32.409	0.783
4096×4096	1114.25	446.462	448.934	422.548	383.159	4.531

Different results have been found between the different compilers at specific optimization levels. Even though the table values are the median of five independent runs, it seems like some variability remains. Likely due to overhead from the operating system, and or different cache behaviors between runs (maybe the L2 and L3 cache fill up and many cache misses are made). This seems to be happening due to non proportional results to the matrix sizes (for example in gcc-10[1], O2 is slower than O1 with a 2048 matrix but significantly faster with a 4096 matrix).

For O0, the results are practically identical between the different compilers, and matrix sizes. I expected this result, as there is no vectorization nor optimization applied to the code, so the compiled binaries should nearly identical. The results are extremely slow compared to the Lapacke implementation and all the optimized binaries, as expected.

For O1, execution times improve considerably, with speedups of approximately ×3.7 for gcc-8 and ×2.5 for gcc-10 and gcc-11. Interestingly, the gcc-8 binaries perform better than those compiled with newer versions.

In the case of O2, gcc-8 and gcc-10 have similar results, improving slightly the performance from the O1 optimization in gcc-8. However, gcc-11 does not seem to benefit from this optimization level, showing equal or slightly worse performance than O1.

In the case of O3, the vectorization should start to be applied (although O3 already can vectorize the code, **-march=native -ftree-vectorize -ftree-slp-vectorize** flags have been added to tell explicitly the compiler to vectorize). The results don't seem to vary from the O2 optimization, likely because not much vectorization is being applied (will be analyzed in the following chapter).

For Ofast, the results in all the compiler versions improve around ~1.1 to ~1.3 times relative to O3. This is an expected improvement, as Ofast enables optimizations which violate requirements of C standard for floating-point semantics[1]. In gcc-11, the performance finally improves compared to previous optimization levels, but still underperforms the other compiler versions.

The results of the optimization levels are similar to my personal expectations for gcc-8 and gcc-10, where each optimization level increase give better results, although I find interesting that no apparent difference is seen between O2 and O3.

The final results situate gcc-8 as the compiler with the best performance, closely followed by gcc-10 and with gcc-11 as last place. This is a very unexpected result, as my personal expectation was either that all the versions would have similar results (as happens with gcc-8 and gcc-10), or that the results would improve slightly when upgrading to more recent gcc versions. The specific case of gcc-11 is quite strange and further analysis with gdb of the assembly code must be done in Task 3 in order to obtain conclusions of the differences between optimization levels in this version.

I find unlikely that the cache behavior varies significantly between different optimization levels, due to the small improvements between the levels. Further analysis on this end will be done in Task 3 to obtain more data around this topic.

---

1 Compiler names have been abbreviated from gcc-8.4.0 gcc-10.1.0 and gcc-11.4.0 to gcc-8, gcc-10 and gcc-11 in order to improve the readability of the report.

In conclusion, the benchmarking results reveal that compiler version differences can have a significant impact on performance, even with the same optimization flags.

Additionally, the variability between runs despite single threaded execution indicates to uncontrollable runtime factors, as NUMA effects, and OS scheduling overhead. These effects become more apparent as execution times decrease with higher optimization levels. Another possible causing factor, could be the overhead from the python script made for benchmarking. This seems unlikely as the overhead should be minimal, and even if it was significant, it would probably apply proportional time increases between executions. The attempt to run a binary multiple times without launching it with python was made and no noticeable differences were found.

Vectorization

Thanks to the script made for compiling, log text files of the compiler vectorization attempts have been created. This chapter of the document will try to analyze why some vectorization oportunities have or haven't been taken by the compiler.

Due to having the best performance, gcc-8 has been selected as the vectorization testing compiler version. O3 has been selected as the optimization level for testing vectorization as it doesn't violate the C standard for floating point semantics.

The summary of the compilation information about the dgesv implementation is the following (the full compilation report can be found in /results/SMALL_compile_logs_rs.log):

1. In the row swapping feature, the first loop for row swapping is vectorized (dgesv.c:37).

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/dgesv.c:37:4:
note: loop vectorized
```

2. The second loop for row swapping is also vectorized (dgesv.c:45):

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/dgesv.c:45:4:
note: loop vectorized
```

3. The loop that applies the multiplication to the a matrix in the first step of Gauss (dgesv.c:72) is vectorized.

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/dgesv.c:72:4:
note: loop vectorized
```

4. The loop that applies the multiplication to the b matrix in the first step of Gauss (dgesv.c:78) is vectorized.

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/dgesv.c:78:4:
note: loop vectorized
```

5. The inner loop that solves the value of each unknown (matrix_utils.h:56) is vectorized.

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/utils/
matrix_utils.h:56:13: note: loop vectorized
```

Those vectorization logs indicate that all the relevant inner loops (outer loops can't be autovectorized), have been vectorized. This contradicts the results from the benchmark, as the results are slightly slower compared to O2, which does not allow vectorization and has been made explicit by using "-fno-tree-vectorize -fno-tree-slp-vectorize".

Further inspection of the assembly code shows that the loops have been partially vectorized (/results/dgesv.S), as in for example this instructions:

```
vmovupd (%r12,%rax), %zmm1
vfnmadd213pd (%rsi,%rax), %zmm2, %zmm1
addq $1, %rdx
vmovupd %zmm1, (%rsi,%rax)
```

Other vectorization logs could give the answer of the contradicting results. In the case of loop **5**, the vectorization logs indicate:

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/utils/
matrix_utils.h:56:13: note: reduc op not supported by target.
```

Which would indicate that the FT3 hardware does not support reduction vector operation and the vectorization has been applied only partially to the loop. The code is being compiled in the FT3 hardware with the -march=native flag, that should tell the compiler the specific hardware the FT3 uses.

The code that uses a reduction seems to be:

```
constant += constant_mul * constant_resolved;
```

An attempt to improve vectorization by changing it to:

```
// Vectorization improvement attempt
double multiplied = constant_mul * constant_resolved;
constant = constant + multiplied;
```

has been made, but seems unlikely that the c compiler does not actually consider this, as it should generate the same operations.

The vectorization logs, also indicate the following in the other loops (**1-4**):

```
/mnt/netapp2/Store_uni/home/ulc/cursos/curso368/HPC_TOOLS/compiler/hpctools/src/dgesv.c:78:4:
note: loop versioned for vectorization because of possible aliasing
```

This indicates that gcc has vectorized the loop but created multiple versions of it, due to not knowing if a and b matrices overlap. The testing to see if they overlap should be determined at runtime by the program, but my personal hypothesis is that it might be failing. The attempt to fix this by telling the compiler that no aliasing is present has been made by using restrict:

```
int my_dgesv(size_t n, size_t nrhs, double *restrict a, double *restrict b)
```

Results of the vectorization

The vectorization attempts made in the loops 1-5 have not been successful and show the same vectorization logs. Why the aliasing note is still appearing is unknown because the restrict keyword should already inform the compiler that no aliasing is present. The compiler logs can be found at results/VECT_1_compile_logs_rs.log.

Another attempt to vectorize the code has been made by using the c array indexers instead of indexing into a double*:

```
int my_dgesv(size_t n, size_t nrhs, double a[restrict n][n], double b[restrict nrhs][nrhs])
```

All the changes made to all the code because of this change can be found at /src_vec, as the /src code was left untouched (the baseline code + row swapping only, without vectorization attempts).

The changes still did not affect the result and the resulting compiler vectorization logs were the same for the 1-5 loops. The compiler logs can be found at results/VECT_2_compile_logs_rs.log.

Conclusions

The benchmarking results show clear performance improvements with higher optimization levels, especially from O1 to Ofast.

The compiler successfully applied autovectorization on the inner loops, as the output assembly shows the vector instructions such as vbroadcastsd and vfnmadd213pd, which operate on multiple double precision elements simultaneously.

However, some parts of the algorithm, seem to remain scalar and limit the overall performance gain.

The use of the restrict qualifier did not help the compiler reduce aliasing nor did indexing the matrices with array indexing.

Overall, the results confirm the impact of compiler optimizations and the importance of memory access patterns for vector and overall performance.

References

[1] “gcc differences between -O3 vs -Ofast optimizations”, [Online]. Available: https://stackoverflow.com/questions/61232427/gcc-differences-between-o3-vs-ofast-optimizations